Computer Engineering and Intelligent Systems
ISSN 2222-1719 (Paper) ISSN 2222-2863 (Online)
Vol 3, No.5, 2012

Improving the Performance of Text Rendering using Font Type Conversion

Zohebahmed Mulla
Dept of ISE R.V.C.E，Bangalore, India
zohebahmedmulla@gmail.com

Raghavendra Prasad S.G
Assistant Professor, Dept of ISE R.V.C.E，Bangalore, India
sgrp.vishnu@gmail.com

Abstract
Printers and PDF are part of our day to day life; enhanced performance of a printer will benefit the users by great deal. Type conversion method proposed to improve performance by overcoming limitation of applications that render characters as shape rather than text which slows down the printer as a PDF file is composed of thousands of character. Rendering of character as shape will increase spool size of the printer since instruction related to every single character construction has to be stored such as number of, path construction commands(moveto, curveto and so on) along with their co-ordinate points. Proposed paper deals with the method to convert subset embedded font in to complete font file say TTF, by adding essential missing data and then rendering them using language framework fonts which results in rendering as text instead of shape, .ultimately reducing the printer spool size and enhances the performance of application.

Keywords- Font type conversion, Font engine, Subset embedded fonts, language frame work font, printer

Introduction

In information technology, glyph is nothing but visual representations of the character. Collection of glyphs constitute a font, a single font might have many versions, such as heavy, medium, oblique, gothic, and regular. These different versions are called faces. All of the faces in a font have a similar typographic design and can be recognized as members of the same family. With the appearance of personal computers, printers and internet at the end of the 20th century, font designers were no longer limited with the specifications of the metal types and started developing fonts which did not adhere to a single well known specification, hence a need for a standardizing font definition was identified by Xerox PARC with development of its first laser printer. Warnock created a language, similar to InterPress, called PostScript to solve the problem. With introduction of Apple LaserWriter first printer to use postscript brought a revolution, both in desktop as well as printer applications. Another kind of printer which was non postscript was developed and Ghostscript was used to print PostScript documents on non-PostScript printers [1].

Performance of Applications is a crucial factor as per user perspective yet, the application fails to address the issue of rendering text in printers. That is current printer renders glyph as shape not as text which increases the spool size and decreases performance. The paper discussed on this front and the mechanism developed to solve the problem using type conversion of native java fonts which are then handled by the java framework itself, solves the problem to a greater extent.

The paper is organized as follows. Section II describes the literature survey on printer classification, working of printer and performance issues in a printer. The solution to the above problem is discussed in section III which speaks about proposed mechanism consisting of converting non proper TTF font to proper TTF, handling of these fonts by java frame work and installation of these fonts temporarily so that other application can use it. Section IV explains about the experimental data collected. Section V is the conclusion and section VI acknowledges the contributions made.

Background

Text rendering is a complex job, in contradiction to many belief that it straight forward. There exist numerous ways to format text within a document, at the same time there might be many paths that yields the same results.

Mapping given text to the final glyph in font requires many transformations, which includes parsing the font data and analyzing language writing patterns. After the final glyph is achieved, it is then followed by rasterizing, merging and filtering so that the final visual must display on viewer or on to printer output. GDI (graphical device interface) and GDI+ are used for this purpose which places rendered text on to the canvas or on to

physical print medium. In this section, the topics discussed would be font classification, subset embedded fonts, printer architecture and control flow as well as the issue which initiated this research and resulted in the mechanism hence proposed.

A Font classification

Bitmap Font: A bitmap font is one that stores each glyph as an array of pixels (i.e a bitmap). It is less commonly known as a raster font. Bitmap fonts are simply collections of raster images of glyphs. For each variant of the font, there is a complete set of glyph images, with each set containing an image for each character. For example, if a font has three sizes, and any combination of bold and italic, then there must be 12 complete sets of images[1][2].

Vector fonts: vector fonts are collections of vector images, i.e. a set of lines and curves to define the border of glyphs. Early vector fonts were used by vector monitors and vector plotters using their own internal fonts, usually with thin single strokes instead of thick outlined glyphs. The advent of desktop publishing brought the need for a universal standard to integrate the graphical user interface of the first Macintosh and laser printers. The term to describe the integration technology was WYSIWYG (What You See Is What You Get). The universal standard was (and still is) Adobe PostScript. Examples are PostScript Type 1 and Type 3 fonts, TrueType and Open Type [1][2].

1. True type Fonts(TTF)

TrueType is an outline font standard developed by Apple as to compete with Adobe's type 1 font in 1980. These fonts are commonly used in Windows and Mac operating system and are based on vector graphics. Strength of true type is it offers the font developer a higher degree of control over font till pixel level.
True type system includes virtual machine which executes programs inside font. Glyphs which are produced with intension that the rasterize produces fewer undesirable features in glyph by taking help of distort and control points within the outline

2. Postscript Fonts

PostScript is an *object-oriented language*, meaning that it treats images, including fonts, as collections of geometrical objects rather than bit maps. PostScript fonts are called *outline fonts* because the outline of each character is defined. They are also called *scalable fonts* because their size can be changed with PostScript commands. Further postscript divided in to type0, type1, type2, type3 and so on.

3. Open type Font

Open type font is a format for scalable computer fonts. It was built on its predecessor TrueType, retaining TrueType's basic structure and adding many intricate data structures for prescribing typographic behaviour. Open Type is a registered trademark of the Microsoft Corporation. The specification germinated at Microsoft, along with contributions from Adobe Systems before it was announced publically in 1996. The specification continues to be developed actively and is presently migrating to an open format.

B Subset embedded font

One while creating a PDF might have come across the option of subset font, wondering what does that option do and does it affect the PDF look. Answer to this question would be straight forward, that it will help decreasing the size of the PDF but it would affect the edit ability of the PDF.For those are neither familiar with this word nor created a PDF, subset embedded font is nothing but embedding only those glyphs or characters that are used in PDF creation. For example, in Helvetica font, if characters from A to F are only used, then those characters will be embedded in the PDF, ignoring the rest.

The file size of the PDF would also be smaller since only a part of the font is being embedded. Editing the file would be possible only if the other user has the same font.
Another option is full font embedding in which complete font is embedded in to the PDF. This increases the file size and allows the other user to edit it even though the font is not installed in the user's system.

C Introduction to flow control in standard printer

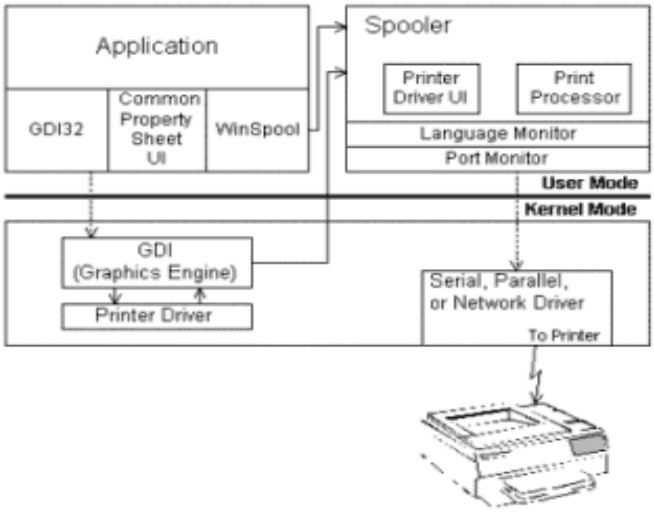

Fig 1: Printer Architecture

(1) Application makes win32 printing and GDI calls; these are primarily functions that are found in Winspool.dll and gdi32.dll. These calls are device independent calls [3].
(2) The Win32 calls are passed to the Kernel - Mode GDI rendering engine. The printer driver in conjunction with the graphics-rendering engine renders the input and passes the raw data on to the spooler [3].
(3) Spooler further processes the data and sends it to the port driver to which the target printer is connected and data is then printed on sheet/document [3].

D Printing issue

As discussed earlier, an application that does not interpret a character as text instead renders as shape resulting in increased spool size as well as making the application non responsive till the printing process is over. Now let us analyze why rendering as shape increases spool size and how to overcome this problem.

Consider a scenario in which a PDF composed of random alphabets and numbers is sent for printing as mentioned above with respect to the flow of control and architecture of a printer. First a GDI call is made and then it is passed to graphics rendering engine.

Graphics rendering engine reads data character by character. In this case, let the first character be A. When character A is read and data is transferred to the spool, character A is not transferred as text instead it is transferred as shape.

For example the raw data at spool for character A is as shown below

co-ordinates- [-22.0, 1357.0, 332.0, 1357.0, 1160.0, 339.0, 1160.0, 1110.0, 1160.0, 1203.0, 1150.0, 1260.0, 1130.0, 1285.0, 1106.0, 1315.0, 1066.0, 1331.0, 1010.0, 1332.0, 962.0, 1332.0, 962.0, 1357.0, 1456.0, 1357.0, 1456.0, 1332.0, 1406.0, 1332.0, 1348.0, 1331.0, 1305.0, 1311.0, 1280.0, 1274.0, 1266.0, 1249.0, 1258.0, 1194.0, 1258.0, 1110.0, 1258.0, -23.0, 1217.0, -23.0, 345.0, 1039.0, 345.0, 247.0, 345.0, 151.0, 355.0, 92.0, 373.0, 71.0, 399.0, 39.0, 440.0, 23.0, 494.0, 25.0, 543.0, 25.0, 543.0, 0.0, 49.0, 0.0, 49.0, 25.0, 97.0, 25.0, 157.0, 23.0, 199.0, 42.0, 224.0, 82.0, 239.0, 105.0, 247.0, 160.0, 247.0, 247.0, 247.0, 1160.0, 207.0, 1206.0, 177.0, 1238.0, 157.0, 1254.0, 137.0, 1269.0, 106.0, 1283.0, 66.0, 1297.0, 48.0, 1303.0, 19.0, 1306.0, -22.0, 1308.0, -22.0, 1357.0]

path construction commands - [moveto, lineto, lineto, lineto, curveto, curveto, lineto, lineto, lineto, lineto, lineto, curveto, curveto, lineto, lineto, lineto, lineto, curveto, curveto, lineto, lineto, lineto, lineto, lineto, curveto, curveto, lineto, curveto, curveto, curveto, lineto, closepath]

Interpretation of the above can be done as command moveto will take two co-ordinate points and will move to that specified position while the command lineto will take two more co-ordinate points and will draw line between the specified points. Similarly, curve to will draw Bezier curve by taking six co-ordinate points and this will continue till closepath is reached. This is how character as a shape is rendered onto physical print media as depicted in the below image.

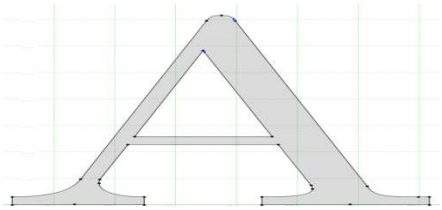

Fig: 2 Construction of character A

Now imagine this PDF made of thousands of characters and if each character has to be rendered as shape, then just imagine spool size due do the data as described above, which increases the time taken to print and ultimately compromises the application's performance. As presented in the experimental result section below file of any size in spool on an average becomes 13 times bigger than the original size. A situation where a file of very large size given as input is rendered as shape, it may not be able to fit into the spool size resulting in crashing or hanging of the printer as well as its application.

To overcome this problem a unique method called font type conversion to produce language framework font is proposed which result in rendering as text instead of shape. Method is described in detail in proposed mechanism section below.

.

Proposed Mechanism

The Font type conversion is driving force behind achieving the goal set at the beginning, type conversion and creation of language framework font is achieved by following three methods.

A Conversion from non-proper TTF stream to Proper TTF stream

Here non-proper TTF is nothing but font files which are not installable. We also consider converting file formats such as PFB, CFF, CID, and OTF so on to proper TTF and the conversion involves adding missing as well some default data to make it a proper TTF stream. Later in same section, we will elaborate how to add missing and default data but first, we will analyze why TTF is chosen over other formats.

Reason for selecting TTF was that it is supported by both dot-net as well as java platforms. Java supports type1 PFB along with TTF while the same is not supported by Dot-net. Another important aspect is that in order to use PFB (printer font binary) stream, you will need PFM (printer font metric) information along with it but PFB stream does not have PFM information along with it, which makes it a highly unattractive option. Based on the two factors and with the intension to make the proposed mechanism to be compatible with dot-net in future, decision of converting to proper TTF was selected.

A hurdle in the process of achieving increased printer performance was to convert non proper TTF which contains glyph data to be extracted from it and combined with some default values to fill up tables in order to convert it to proper TTF files so that, it can be loaded by java and Dot-net frame work. To solve this problem one has to dig deeper in to structure and specification of TTF and other non TTF font files. To keep it simple, we will convert PFB to TTF .The steps are explained in the example below in a simple yet meaningful way.

A PFB can be bifurcated into 3 parts. The first section is composed of font properties such as version, font name, family name and so on while the second section consist of encrypted data which includes glyph information while the last section called the end marker specifies end of file, diagrammatically as depicted below.

.

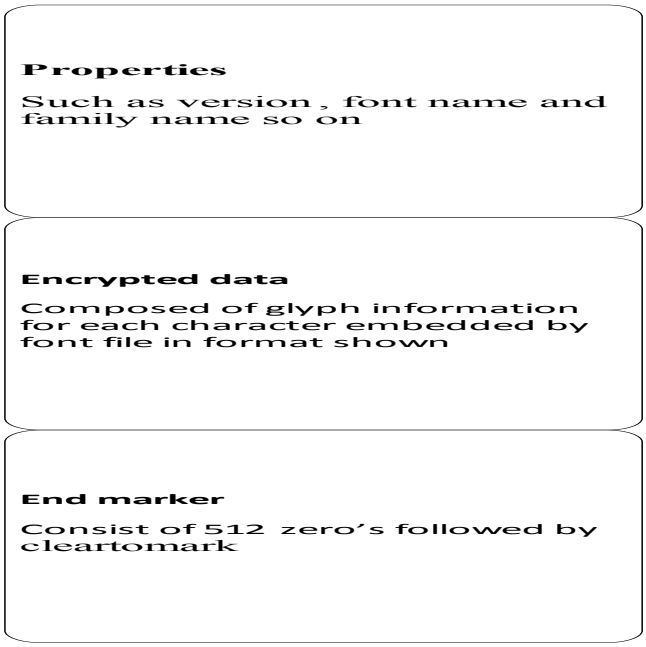

Fig: 3 structure of PFB

Similarly TTF consist of Table Directory, Required and Optional Tables where Table directory begins at byte 0 with the Offset Table and it contains information such as version number, number of tables, search range, and so on followed by at 12 byte there will information such as tag, offset to tables, checksum and length.

While requirement table consist of tables such as cmap, glyf, head, hhea, hmtx, loca, maxp, name, post and OS/2. Out of which glyph table is of primary importance to us as the byte array obtained from the PFB has to be imported in TTF glyph section.

Last section of optional tables consists of around 14 tables out of which an attempt was made to remove as many tables as possible or add default values to the tables to make the conversion much simpler. Below is diagrammatic representation of the TTF structure

Fig 4: structure of TTF

Now that the structures of both are known, let us see how the information about properties and glyphs such as byte array of co-ordinates and path construction commands will be filled in TTF.

First properties such as FontBBox, version, font name family and so on are read from PFB and then the values such as FontBBox is then put into head table. Similarly family, font name and copy right extracted from PFB are then substituted into the name table of TTF and so on. The process goes on till the required properties have been extracted from PFB and substituted into TTF.

The second important aspect of this conversion is substituting glyph information extracted from PFB in to TTF. For this, we have to construct a glyf table where the path construction commands of PFB goes into instruction[ ]. One of the table variable of data type byte contains the path construction commands of every glyph and co-ordinates is then put into xCo-ordinates[ ] , yCo-ordinates[ ] of the glyf table. Along with this table other tables such as maxp, head, post and loca have to be generated by either taking the default specific values or extracting them from PFB to make a proper TTF tables. The above mentioned steps are represented diagrammatically as follows,

.

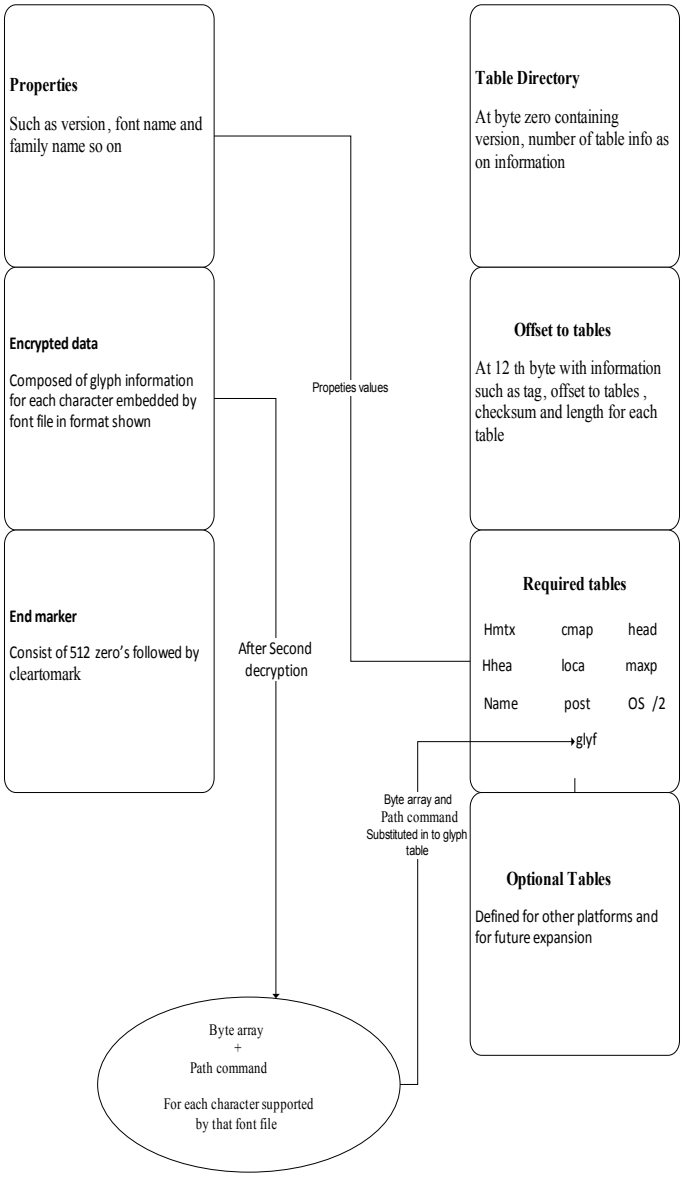

Fig 5: Conversion of non proper TTF to proper TTF

B Native java fonts handling by java framework

In the previous step, we have created a proper TTF stream from non proper TTF or non TTF font stream. This results in creation of java fonts which helps in drawing on the canvas as text rather than shape.
For this purpose we use an API called **graphics.drawString** method, which draws the specified text string at the specified location with the specified Brush and Font objects.

The API is as explained below,

public abstract void drawString(String str, float x, float y)

This API renders the text specified by the specified String, using the current text attribute state in the Graphics2D context. The baseline of the first character is at position (x, y) in the User Space. The rendering attributes applied include the Clip, Transform, Paint, Font and Composite attributes. For characters in script systems such as Hebrew and Arabic, the glyphs can be rendered from right to left, in which case the co-ordinates supplied is the location of the leftmost character on the baseline.

Parameters:
str - the String to be rendered
x - the x co-ordinate of the location where the String should be rendered
y - the y co-ordinate of the location where the String should be rendered

C Temporary installation of fonts

Main reason for installing font is that it can used by other devices(printer) or application can use the installed fonts in private collection folder, as it will decreases manual work that is required if font is not installed under such circumstances whole font file have to copied which is not good approach for programming.

But one cannot install font permanently in others windows font directory due to following reason
(1) User may not want anyone to install any font in his system without his knowledge.
(2) One may not have required permission to install fonts directly in window's font directory.
For above specified reasons we install font temporarily in private font collection instead of windows font directory. Once the object using it is killed/terminated the temporally installed fonts are also removed from the system automatically.
JNA (java native access) library is used
And API called **AddFontResourceEx [6]** which is of the format as shown below

```
int AddFontResourceEx(
  __in  LPCTSTR lpszFilename,
  __in  DWORD fl,
  __in  PVOID pdv
);
```

Were parameter lpszFilename can take file with extension such as .TTF,.OTF,.MM and so on

Once the three steps are completed its results in rendering as text rather than shape since we send co-ordinate values and commands to the printer. Else without proposed method, the printer font processing engine has to convert the glyph in to corresponding co-ordinate points and commands to draw onto the output medium which is nothing but rendering as shape.

Using Experimental Result and Analysis of Performance

Postscript and non postscript based printer which takes PDF for printing purpose render text as shape which results in increased spool size of printer as a PDF will be composed of thousands of characters. Now consider a scenario where one printer in connected to many systems through LAN where it may receive multiple requests at once. Under these conditions, the printer performance is crucial to successfully complete all the requests in considerably less amount of time.

In this paper we have taken two criteria namely Spool size and output file to compare the performance of printer before and after the proposed mechanism is implemented.

4.1 comparisons with respect to spool size

Let us consider a sample file of size **856kb** as an input to the printer. If rendered as shape, the spool size will become **18.3Mb.** This is an increase of 17780kb, which is approximately 21 times the original size, the reason for which is explained in related work section. If the same file is rendered as text with the help of proposed mechanism of font type conversion, the spool size will be **8.02Mb** which is an increase of 7540 kb in spool size or an **8** times increase as compared to original size of file, and improvement of around **13 times** as compared to previous method.
Similarly experiments were carried on files of size .91, 1.83, 3.7 and 4.44 mb respectively. Files and spool size observed were 37.5, 78.2, 151 and 301 all in mb, if rendered as shape. If they were rendered as text, the spool size observed was 7.14, 15, 28.6,56, 110 mb respectively with an overall reduction of 63.24 percent in the spool size when compared to rendering as shape.

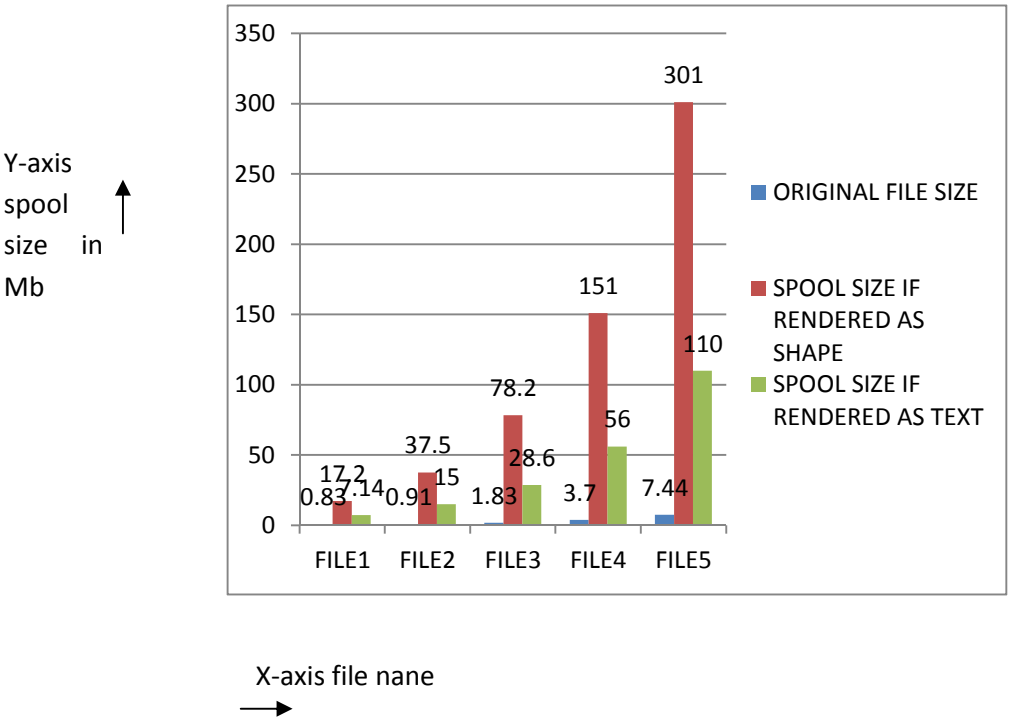

Fig 6: graph showing spool size

4.2 Proof of concept

After identify the problem in printing and proposing a method to solve it, now comes how to show that aim has been achieved. That is how to show that output is text not shape for this purpose a test was done in which a virtual printer was taken and content were then printed on to PDF, if PDF created by this way had subset embedded fonts in the properties section that will prove that the output was generated as text if there no embedded fonts in the PDF then output in inform of shape.

Our experimental results have proved that data which was printed on to PDF had subset embedded font in it, hence this proof that proposed concept works as well as spool size is reduced thereby improving the performance

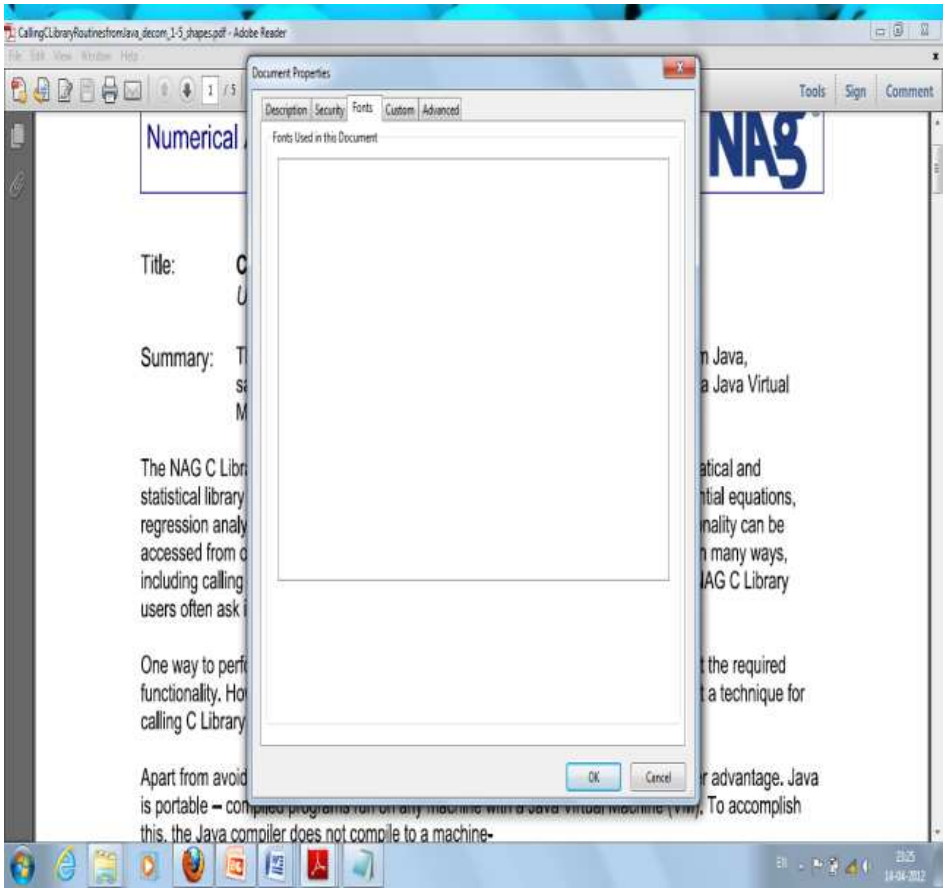

Fig 7: PDF font properties is blank if rendered as shape

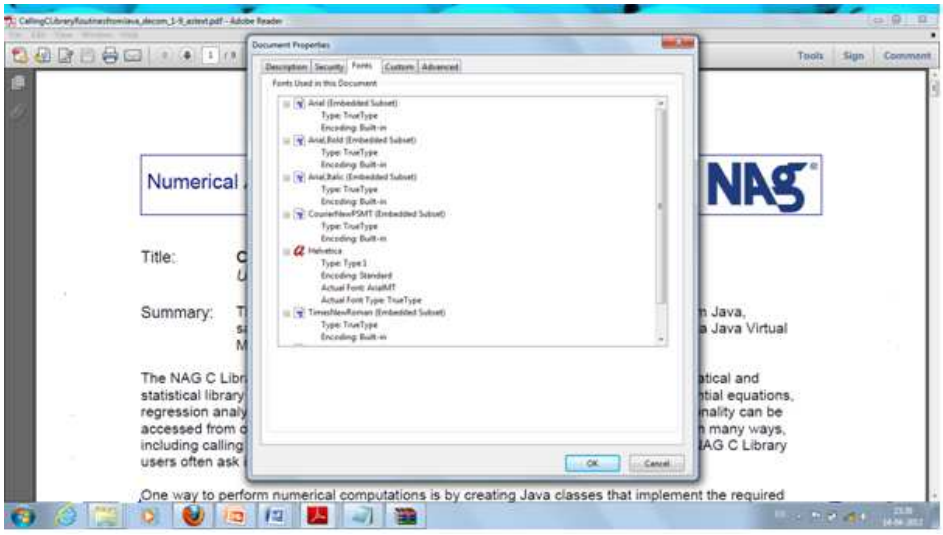

Fig 8: PDF font properties is not blank if rendered as text

Conclusion

In this paper we proposed a method to overcome the limitation of application which do not render character as text for subset embedded fonts in PDF and hence to improve the performance.

[1] Reducing the size of file in printer spool by creating native font that can be loaded by java frame work.

[2] Proposed method is easy to implement at same time quite efficient since once non proper ttf stream is converted in to proper ttf stream by using font type conversion and we get , language framework font which are then handled by java framework using inbuilt API's.

[3] Improves the response time of application. By reducing the time for printing. Since it made non-responsive till printing job is done.

There are still a great many works need to be studied in the future with respect to font processors and rendering issues

Acknowledgment

Although much of the conceptual work and initial software for this was done as part of the Xtreme font engine program. We wish to thank Girish Patil, Shivaranjini, Santhanam, Snehashree, Abhishek and Krishna Kumar from Gnostice Information Technology for kind help and support

References

[1] The WIKIPEDIA website. Available. [Online]. http://en.wikipedia.org/wiki/PostScript.

[2] Dipali B Choudhary, Sagar A Tamhane, Rushikesh K Joshi "A SURVEY OF FONTS AND ENCODINGS FOR INDIAN LANGUAGE SCRIPT" International Conference on Multimedia and Design (ICMD), 23-25 sept 2002, Bombay.

[3] The Calsoft Lab website, Available [online] http://www.calsoftlabs.com/whitepapers/bitmap-driver.html.

[4]Microsoft topology, website Available [online]
http://www.microsoft.com/typography/SpecificationsOverview.mspx

[5]Graphics 2D in java website, Available [online]
mhttp://download.java.net/jdk7/archive/b123/docs/api/java/awt/Graphics2D.html

[6] Microsoft MSDN website, Available [online]
http://msdn.microsoft.com/enus/library/dd183326%28v=vs.85%29.aspx.

[7] Thomas w phinny "TrueType, PostScript Type 1,& OpenType: What's the Difference?", white paper December 2004 by adobe